Handwritten Digit Classication using 8-bit Floating Point based Convolutional Neural Networks

Gallus, Michal; Nannarelli, Alberto

Publication date:
2018

Document Version
Publisher's PDF, also known as Version of record

Handwritten Digit Classification using 8-bit Floating Point based Convolutional Neural Networks

Michal Gallus and Alberto Nannarelli (supervisor)
Danmarks Tekniske Universitet
Lyngby, Denmark
s172679@student.dtu.dk

***Abstract*—Training of deep neural networks is often constrained by the available memory and computational power. This often causes it to run for weeks even when the underlying platform is employed with multiple GPUs. In order to speed up the training and reduce space complexity the paper presents an approach of using reduced precision (8-bit) floating points for training hand-written characters classifier LeNeT-5 which allows for achieving 97.10% (Top-1 and Top-5) accuracy while reducing the overall space complexity by 75% in comparison to a model using single precision floating points.**

***Keywords*—approximate computing, deep learning**

I. Introduction

Image classification has recently been dominated by Convolutional Neural Networks (**CNN**s) that provide state of the art accuracy, sometimes reaching human level capabilities. In order to achieve better results by means of accuracy, the deep neural network models rapidly grow in size, resulting in long training times, high demand for memory and computational power. One of the most commonly seen solutions to this problem concerns employing large CPU Clusters [1], or Graphical Processing Units (**GPU**s) [2] which recent models of started including deep learning specific hardware units, such as Tensor Cores in Nvidia Volta [3]. Application Specific Integrated Circuits (**ASIC**s) as well as FPGA solutions have also been proposed [4].

What the majority of aforementioned solutions fail to exploit is the inherent resilience to errors associated with deep learning applications. Because the neural networks are exposed to work with real-life noisy data while maintaining successful results [5], they do not require full precision range provided by the hardware in a form of 64-, 32- and even 16-bit floating point numbers. Limiting the precision of underlying primitives which build up the network can result in power, time and memory savings.

Nearly all of the state of the art convolutional neural network models spend most of the processing time on convolution and max pooling layer [6], [7], [8]. Timespan of performing all convolution operations on the whole network amounts to 80% of total time spent on training [9]. At the same time, nearly 20% of remaining time is spent on subsampling, also known as max pooling operations. These on the other hand are memory bound, and their speed is limited by the CPU's cache size and bandwidth of the bus.

This work was done as part of the coursework in "Design of Arithmetic Processors" under the supervision of A. Nannarelli.

In order to address this problem, this paper proposes usage of 8-bit floating point instead of single precision floating point which allows to save 75% space for all trainable parameters, and possibly reduce power and computation time. To test the proposed solution the computations are ran on LeNeT-5 [10] network using MNIST hand-written images dataset [11].

The paper is organised in the following manner: Section II presents the current advancements in the field. Section III concerns the main topic of the paper where the structure of 8- and 12-bit float will be presented along with the architecture known as LeNeT-5. In section IV reader can find the results of training and validation of the aforementioned network using floats of different bit-widths. Finally the section V concludes the paper and proposes further improvements.

II. Related work

During the design of special purpose hardware intended for deep neural networks processing, one of the key factors to consider is the used precision and format of numbers used for storing the results and performing the computations. Most of the literature so far has focused on using limited precision during the forward pass (also known as inference), while training the network on single precision. Some of the recent works have presented usage of custom hardware implementations [12], while others have employed the usage of FPGAs [4] for the sake of reconfigurability or using SSE3 vector instructions to perform multiple (up to 16) 8-bit operations in parallel [13].

Other studies concern usage of different number format. In [14] 24-bit floating point processing units are used, while [15] uses 16-bit fixed point with stochastic rounding, achieving errors of 0.70% on modified LeNeT-5 architecture. This result has later been reproduced along with introduction of dynamic fixed point by [16]. Hammerstrom [17] shows successful training of the network using 16- and 8-bit fixed-point arithmetic.

This work presents that both training and inference is possible using not only fixed- but also low precision (down to 8-bits) floating point arithmetic. This includes direct application to not only simple perceptron networks, but also models of higher complexity, such as LeNeT-5 architecture which can be used as fully functional module for Optical Character Recognition (**OCR**) program.

III. Training low precision deep neural network

Typical implementations of deep neural networks using gradient descent training method use 32-bit floating point

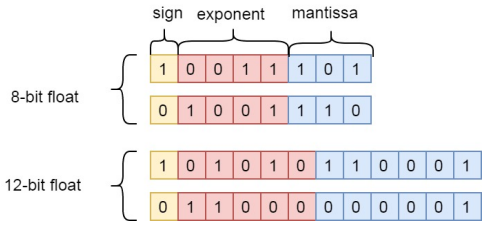

Fig. 1. Representation of low precision floating points used

representation of real numbers. In order to prove that such high precision is not necessary for successful training and inference, the complete from-scratch C++ implementation of LeNeT-5 has been created that would easily allow for switching between underlying data type that is used for storing image data, weights, and all intermediaries. All the operations performed on the data use single-precision floating point arithmetic. Therefore the type representing floating point of variable precision had to have its conversion operators from- and to-32-bit float defined. These, along with other details on implementation have been presented in the following section.

A. Low-precision floating point

Floating point format is an industry standard of representing real numbers. A single number consists of a sign, exponent and finally a mantissa, all portrayed in the Figure 1. Exponent is used to allow for representing a wide range of the real numbers, while mantissa serves for assuring adequate precision. A real number computed from its floating point format can be computed based on the following formula:

$$value = (-1)^{sign} \times (1 + \frac{mantissa}{2^M}) \times 2^{exponent - bias}$$

where M is the amount of bits used to represent mantissa, while $bias$ is equal to $2^{E-1} - 1$, E being the number of bits used for exponent.

Bit-width	Exponent bits	Mantisa bits
32	8	23
16	5	10
12	5	6
8	4	3

TABLE I. PRECISION USED DURING TRAINING AND INFERENCE

Table I shows all four representations that have been subject to neural network training and testing during the experiment.

B. Implementation

Since the current CPUs do not support floats of bit-width lower than 32-bits, the need for implementing the low-precision primitives has arised. This could potentially include building a class representing a minifloat and including all the arithmetic operations on the software level. This however, would incur a very high overhead, which along with typically long neural network training times would result in unacceptable amount of days and hours spent on debugging and testing. Therefore, the decision has been made to develop a primitive that would only store the value of low-precision float, and define conversions from and to a single precision float. The conversions are needed, so that all the arithmetic operations can be performed on primitives that can benefit from hardware acceleration.

The following subsections cover how the conversions are being handled. Please note, that from now on, the notion of **minifloat** will be used to denote a non-standard low precision floating point number such as 8- or 12-bit float.

C. Float → minifloat conversion

In ordert to convert 32-bit floating point number to minifloat, the following steps need to be performed:

1) Determine and set the sign bit
2) Compute rounding of mantissa
   a) Determine guard, last, round and sticky bits
   b) Compute $G(L + R + T)$ and save result to r
3) Shift float mantissa right by $(23 - S)$ and save to m
4) Save the minifloat mantissa m_m as a sum of obtained shift m and the rounding r
5) Determine if overflow has occured using equation $o = r \cdot \overline{m}$
6) Extract the float exponent e_f and subtract the float bias $e_r = e_f - (2^7 - 1)$ to obtain the raw exponent e_r
7) Compute minifloat exponent e_m by adding the minifloat bias and an mantissa overflow flag $e_m = e_r + (2^{E-1} - 1) + o$
8) If $e_m < 0$:
   a) Determine the value of the guard bit (the most significant bit that will be truncated) and store it in G_u
   b) Compute the new mantissa using equation $(m_m + 2^S) \cdot 2^{e_m} + G_u$ where 2^S is the hidden bit, 2^{e_m} shifts the value right by an absolute value of minifloat exponent, and addition of G_u approximates rounding
   c) Set minifloat exponent e_m to zero

   Else if $e_m > 2^E - 1$:
   a) Based on the sign, set the number to positive or negative infinity, by setting exponent to $(2^E - 1)$ and mantissa to 0

Where S and E are the bit lengths of minifloat significant and exponent respectively.

D. Minifloat → float conversion

Conversion between minifloat and single precision float can be summed up in the following algorithm:

1) Check if the minifloat represents an infinity, and if it does, return the float version of it
2) If the exponent $e_m = 0$ (subnormal):
   a) If the mantissa $m_m = 0$ return 0
   b) Else, determine the position of leading one in mantissa, and save it to p
   c) Set float mantissa m_f to m_m shifted left by $(23 - p)$ bits using equation $m_m \cdot 2^{23-p}$
   d) Calculate raw exponent using equation $e_r = -p - (2^{E-1} - 1)$
3) Else:

a) Shift minifloat mantissa left by $(23 - S)$, using formula $m_f = m_m \cdot 2^{23-S}$
b) Calculate raw exponent by subtracting the bias using equation $e_r = e_m - (2^{E-1} - 1)$

4) Compute float exponent by adding bias: $e_f = e_r + 2^{E-1} - 1$
5) Determine and save the sign.

E. MNIST dataset

The dataset used for the training has been chosen to be the collection of handwritten digits, collected and published by Lecun in [11]. Sample of examined data has been shown in Figure 2. The set consists of 60,000 training images and 10,000 validation images. Each image has a size of 28×28 and represents a digit from 0 to 9. Pixel values have been normalized to lay in $[0, 1]$ range. The data did not undergo any augmentation.

Fig. 2. Visualisation of small subset of MNIST - the training dataset

F. LeNeT-5 Architecture

In order to test the influence of the floating point precision on the training and inference of recognition of handwritten characters, the architecture proposed by Lecun in [10] called LeNeT-5 is used.

The network consists of the following layers:

1) Input Layer
2) Convolution 5×5, 6 feature maps
3) MaxPooling 2×2
4) Convolution 5×5, 16 feature maps
5) MaxPooling 2×2
6) Convolution 5×5, 120 feature maps
7) Fully Connected, 10 feature maps

The amount of trainable parameters, amounts to 60500. From the most common activation functions such as ReLU, sigmoid, the hyperbolic tangent has been chosen to activate feature maps for both convolutional and fully connected layers. The network doesn't use any special learning improvements such as weight decay which would slightly decrease weights magnitude after each epoch or learning rate decay which would do similar, but with respect to the learning rate parameter. Instead it maintains a constant learning rate of 0.0015. The loss function used is a Mean-Square-Error (**MSE**).

IV. Results

The results of running simulation on different precisions presented in Table II show that there is a very low (2.06%) loss of accuracy between usage of 8- and 32-bit floating points. On the other hand, 12-bit floating points show very little performance degradation (by 0.63%). At the same time the 16-bit format preserves the same accuracy as single precision format, and as displayed in Figure 3 it has a comparable to 32-bit speed of convergence.

8-bit float	12-bit float	16-bit float	32-bit float
97.11%	98.63%	99.18%	99.17%

TABLE II. Highest accuracy achieved during the training

If the application in question can sacrifice the 2% of accuracy, it'll allow for overall memory reduction by a factor of **4**. This implication associates with itself several consequences. First of all, when used in CPU with embedded on-chip cache memory, it allows for storing **4x** as many values as in case of single precision float. This is important especially for subsampling layers which are memory bound. Moreover, since in some cases CNNs are used in real-time systems such as i. e. Autonomous driving cars, usage of low-precision floats would loosen the bandwidth requirements, or allow for sending images at greater resolution than in case of 32-bits, or sending image frames at higher rate.

Furthermore, an arithmetic unit operating on the 8-bit floating point can be proposed. Such unit would cover lower amount of area compared to ordinary 32-bit floating point ALU. Lower amount of bits decreases carry propagation time, as well as number of partial products computed during multiplication, so often used when computing convolutions. Due to reduced area, more of such units could be embedded on the chips, resulting in greater parallelism.

Alternatively, since the smaller-area units use lower amount of energy, these could be used in mobile device CPUs, and continue the trend of off-loading simple recognition tasks from cloud-computing to embedded systems.

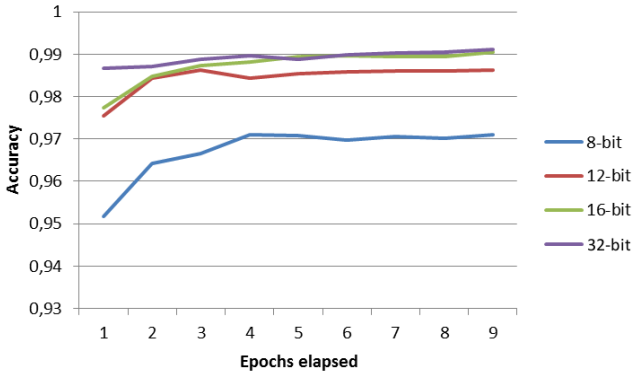

Fig. 3. Plot of accuracy vs. epochs elapsed during the network training

V. Summary an future works

This paper has shown that training and inference of deep neural networks can be conducted using very low-precision

floating point primitives. This can be exploited to significantly optimize memory usage (up to **4** times) and to design either a power- or compute-efficient hardware dedicated for deep learning purposes.

Nevertheless, the increase of 2% in error classification when 8-bit floats are used can be unacceptable for accuracy-critical applications. Moreover, the hardware used for rounding implementation is crucial for retaining utmost of number's identity, yet expensive, therefore some future works could concentrate on implementation of simpler rounding schemes than those presented in the paper.

Potentially, results of this work could be extended to examine and cover usage of low-precision floating-point numbers in different than image classification deep learning applications.

REFERENCES

[1] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

[2] A. Coates, P. Baumstarck, Q. Le, and A. Y. Ng, "Scalable learning for object detection with gpu hardware," in *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*. IEEE, 2009, pp. 4287–4293.

[3] I. Volta, "The worlds most advanced data center gpu," *URL https://devblogs. nvidia. com/parallelforall/inside-volta*, 2017.

[4] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "Neuflow: A runtime reconfigurable dataflow processor for vision," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*. IEEE, 2011, pp. 109–116.

[5] D. Rolnick, A. Veit, S. Belongie, and N. Shavit, "Deep learning is robust to massive label noise," *arXiv preprint arXiv:1705.10694*, 2017

[6] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[8] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[9] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *International conference on artificial neural networks*. Springer, 2014, pp. 281–290.

[10] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[11] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[12] J. Kim, K. Hwang, and W. Sung, "X1000 real-time phoneme recognition vlsi using feed-forward deep neural networks," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*. IEEE, 2014, pp. 7510–7514.

[13] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," in *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, vol. 1, 2011, p. 4.

[14] A. Iwata, Y. Yoshida, S. Matsuda, Y. Sato, and N. Suzumura, "An artificial neural network accelerator using general purpose 24 bits floating point digital signal processors," in *IJCNN*, vol. 2, 1989, pp. 171–182.

[15] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, 2015, pp. 1737–1746.

[16] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," *arXiv preprint arXiv:1412.7024*, 2014.

[17] D. Hammerstrom, "A vlsi architecture for high-performance, low-cost, on-chip learning," in *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*. IEEE, 1990, pp. 537–544.